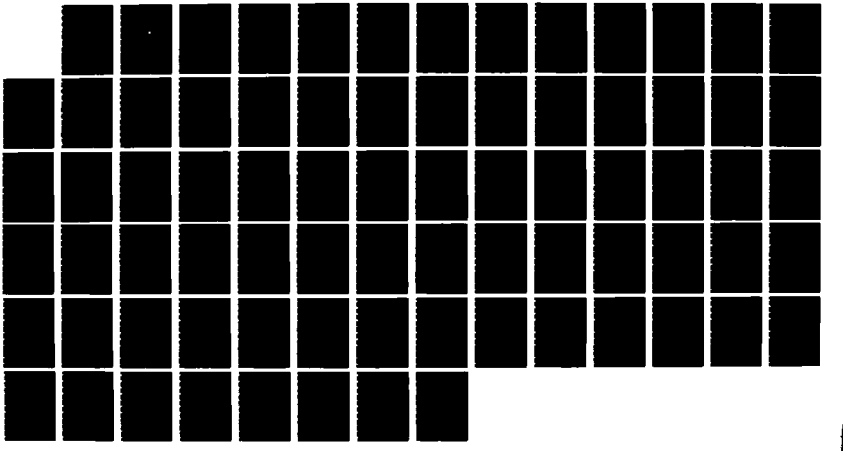
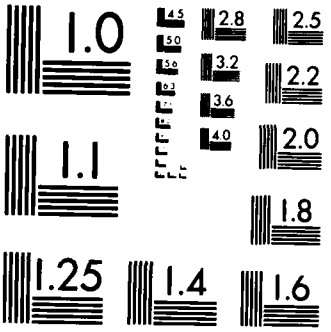
AD-A171 773 FOURTH GENERATION PROGRAMMING LANGUAGES(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA E L JACOBSON JUN 86 1/1

UNCLASSIFIED F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

FOURTH GENERATION PROGRAMMING LANGUAGES

by

Everett Lee Jacobson

June 1986

Thesis Advisor: C. Thomas Wu

Approved for public release; distribution is unlimited

SECURITY CLASSIFICATION OF THIS PAGE AD-A171773

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b DECLASSIFICATION / DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b. OFFICE SYMBOL (If applicable) Code 52	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000	7b. ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000
8a NAME OF FUNDING / SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO.

11 TITLE (Include Security Classification)
FOURTH GENERATION PROGRAMMING LANGUAGES

12 PERSONAL AUTHOR(S)
Jacobson, Everett Lee

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Master's thesis	FROM _____ TO _____	1986 June	75

16 SUPPLEMENTARY NOTATION

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Fourth Generation

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

With an ever increasing demand for new program applications and the failure of older generations of languages, such as COBOL, PL/I, PASCAL, etc., to keep up with this increased demand, there exists a need for new techniques and approaches to programming. Greater programmer/user productivity and enhanced user friendliness, to allow more end users to develop applications on their own, are goals sought by industry in order to reduce skyrocketing backlogs of applications. This paper describes a new generation of programming languages, used in the development of business and scientific applications, that addresses and achieves these goals. The basic characteristics of Fourth Generation Languages is reviewed and the design and implementation of a Fourth Generation Language is proposed. Although Fourth Generation Languages do increase user productivity and are easier to learn and use than previous generations of

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION Unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL C. Thomas Wu	22b TELEPHONE (Include Area Code) (408) 646-3391 22c OFFICE SYMBOL Code 52Wq

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

19. ABSTRACT (cont'd)

languages, much research remains to be done before general end user computing becomes the norm rather than the exception.

Approved for public release; distribution is unlimited.

Fourth Generation Programming Languages

by

Everett Lee Jacobson
Captain, United States Marine Corps
B.S., Washington State University, 1976

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1986

Author: _____
Everett Lee Jacobson

Approved by: _____
C. Thomas Wu, Thesis Advisor

_____
Bruce J. MacLennan, Second Reader

_____
Vincent Y. Lum, Chairman
Department of Computer Science

_____
Kneale T. Marshall,
Dean of Information and Policy Sciences

ABSTRACT

With an ever increasing demand for new program applications and the failure of older generations of languages, such as COBOL, PL/I, PASCAL, etc., to keep up with this increased demand, there exists a need for new techniques and approaches to programming. Greater programmer/user productivity and enhanced user friendliness, to allow more end users to develop applications on their own, are goals sought by industry in order to reduce skyrocketing backlogs of applications. This paper describes a new generation of programming languages, used in the development of business and scientific applications, that addresses and achieves these goals. The basic characteristics of Fourth Generation Languages is reviewed and the design and implementation of a Fourth Generation Language is proposed. Although Fourth Generation Languages do increase user productivity and are easier to learn and use than previous generations of languages, much research remains to be done before general end user computing becomes the norm rather than the exception.

TABLE OF CONTENTS

I. INTRODUCTION

A treatise on computer programming languages should begin with asking why we even have or need these languages. The answer is simply that programming languages are the medium of communication between man and machines, enabling us to instruct the machine as to what we want it to do. Two of the primary goals in the development of programming languages have been to make them easy to use yet rich in functionality or powerful enough to accomplish the wide range of tasks various users might demand of their computing system. Or another way of looking at these goals is that we continue to seek ways to make computers more accessible or useful to more people while simultaneously improving their productivity in the development of application programs. This paper will explore the advances made toward meeting these goals, summarize the basic characteristics common to all languages considered to be Fourth Generation Languages and propose a language to achieve this goal.

A. BACKGROUND

Currently, we have advanced through three generations of languages and a new generation is under development, although some would argue that this fourth generation of languages is already present on the market. This moot point will be settled with time and by those with the advantage of hindsight.

However, I believe that most people would agree that the first generation of programming languages began with the advent of the computer itself and comprised machine languages. Unique to each computer model, machine languages are based on sequences of zeroes and ones, a code quite easily followed by machines but tough for the human eye and mind to keep track of and program in. This difficulty of dealing with zeroes and ones was overcome by the development of mnemonic codes which could be translated into zeroes and ones for the machine but which could be much more easily understood by programmers--these were called assembly languages. But assembly languages have also proven to be rather difficult to use and understand even in simple computing tasks. So, yet another generation of languages, the third, was spawned and they (PL/I, COBOL, FORTRAN, etc.) have proved to be easier to use and understand because of the structured coding and procedural approach to problem solving utilized by these languages. Can we than say that these Third Generation Languages have achieved the goals of being easy to use and functionally powerful? The answer lies with those who are using these programming languages--to a programmer with excellent training and years of experience, any or all of these languages may be considered easy to use but to someone with little or no programming experience, they are extremely confusing to say the least.

B. THE CHALLENGE

Since the marketing of the first computer, the usefulness of computers has increased substantially due to advances in both hardware and software, and the number and experience level of people utilizing computers has also increased. Simultaneously, computing costs have declined, primarily due to hardware. I am convinced that more and more people will be using or wanting to use computers in their daily lives and unless better programming languages are developed, many of these people will face seemingly insurmountable hurdles because they just do not have the time or inclination to learn a programming language. As a result they will continue to rely on overloaded data processing centers and "canned" software which may or may not meet their current needs and most likely not meet their future needs. Those who wish to use the computer to do something beyond the capabilities of the "canned" programs will discover that they have to write their own program. Learning to program, unfortunately, is still a time consuming and oftentimes frustrating endeavor.

The challenge is to bring computer capabilities usefully and simply to people whose work can be benefitted by programming (Shu, 1985, p. 326). In other words, I believe the Third Generation Languages have not gone far enough in making communication with a computer easy as well as functional. This is where the Fourth Generation Languages must continue

the advance toward better man and machine communications, and make computer usage truly acceptable to the general populace.

II. WHY FOURTH GENERATION?

Over the past couple of decades, programming languages have evolved rather slowly. Although there have been differences in data, control, name and syntactic structures, the basic constructs of FORTRAN, COBOL, PL/I, PASCAL, etc., remain somewhat similar. Dramatic increases in programmer productivity have not been forthcoming, yet the need for improvement in this area continues to grow. James Martin (1985, p. 1) highlights this need as follows: "By any set of estimates of future computing power, the productivity of application development must increase by two orders of magnitude over the next ten years. This cannot happen if computers are programmed with COBOL, PL/I, PASCAL, or ADA." And as the use of computers continues to spread, many people who are not experienced programmers must be able to put computers to work for themselves and their employers. Application development without professional programmers is becoming a vigorous trend in computing (Martin, 1982, pp. 56-58). Application programs will increasingly be created by end users, business consultants and system analysts. These individuals need a powerful language with which they can quickly build their own applications and a language which allows them to be able to concentrate on the application rather than on the intricacies of coding. Removing

unnecessary complexity is very important because it allows the user to spend his mental effort on what really matters--the purpose of the application. End users require languages that are as easy to use as possible; user friendly languages without the need to remember mnemonics, formats, sequences and complex controls.

All of the previous generations of programming languages have not satisfied these needs. Therefore, a new generation of computer languages needs to be developed which is more powerful than the previous generation so that results can be obtained much faster. It must also address the issue of user friendliness. Currently, new tools referred to as "High-productivity languages," "Non-procedural languages," and "Application generators," are being introduced and are providing us with an improved ability to put the computer to use. Each of these tools fit or come close to fitting the criteria outlined in the next section and thus may be classified as Fourth Generation Languages. With Fourth Generation Languages, computer power is becoming available to any thinking person and without the need of extensive training in programming.

Most Third Generation Languages utilized a monologue approach, with the user writing a program, compiling it, debugging it and finally running the program. However, many of the Fourth Generation Languages employ a dialogue, with the user and computer interacting as the specific

application is being built. Dialogues, where the user may respond to menus, fill in panels presented to him on the screen, move a cursor or bar on a screen or manipulate data in screen windows, allows the user to catch errors while they are being made or avoid them altogether so that later debugging is simplified.

Fourth Generation Languages are clearly distinct from the previous generations of programming languages and these distinctions are what give these languages the edge in terms of enhanced productivity and increased user friendliness. Their impact will shape data processing developments in the rest of the decade and beyond.

With simple commercial data processing applications, improvements in productivity due to the use of a Fourth Generation Language, vice a Third Generation Language, have been documented to be as high as 80 to 1. However, a 10 to 1 improvement is more typical (Martin, 1985, pp. 75-77). In order to have a major impact on programmer productivity, though, a Fourth Generation Language needs to be more than just another language which affects only the programming part of the life cycle of a program/system under development. It must include prototyping, iterative design and tools which concentrate on the specification and design phases of life cycle development, as well as be capable of generating code directly from the design. Some Fourth Generation Languages are capable of enhancing this "front-end" of system life cycles but much more research is needed.

Although Fourth Generation Languages seem to offer hope in the areas of improved productivity and easier program development, problems still remain. For instance, some Fourth Generation Languages are very limited in what they can actually develop and when applied to an inappropriate problem, they can cause more problems or fail to achieve the required results. This inflexibility in application has made many wonder if Fourth Generation Languages are really an improvement. "Some Fourth Generation Languages are over-sold and do not have the capabilities needed for complex systems." (Martin, 1985, p. 79) Others claim to be easy to learn and use, yet overwhelm most users. Some Fourth Generation Languages which have proven very effective when working with databases, make no provision for application development and thus are really only glorified database management systems. The volume of languages which claim to be Fourth Generation continues to increase. Therefore, it requires considerable skill to select a Fourth Generation Language from the bevy of candidates, to ensure that it is suited to your simple, as well as complex applications and to check that it provides adequate machine performance. In general, a high level of professionalism is needed in the control and management of user computing, especially since the advent and proliferation of Fourth Generation Languages.

III. FOURTH GENERATION LANGUAGE CRITERIA

In order to distinguish this new generation of programming languages from others, criteria or goals must be established to evaluate specific languages and determine where they fit into the family of programming languages. The great variety and diverse scope of these new languages is fantastic. Some are powerful, concise and thereby cryptic at first sight. Some have excellent graphics and are easy to use but lack flexibility. Some are suitable for end users with little training; some are intended for professionals. Some are closely linked to their own database software; some will operate on many types of files. One reason for this paper is to clarify what is meant by Fourth Generation Language and possibly bring a little order to the chaotic and boiling morass of originality and new ideas found in the field of computer languages. Accordingly, the following discussion contains criteria/characteristics which a language must possess to be considered a Fourth Generation Language. And with these characteristics, large improvements in user productivity and increased user friendliness of the language appears not only possible, but hopefully, inevitable.

A. EASY TO LEARN

For a language to be easy tc learn, it should be simple, natural, logical and have few things to memorize and be

based on or related to something already well known by the general populace (the eventual users of the Fourth Generation Languages). By simple, I mean understandable, basic or fundamental and lucidly explained through a concise description and a few examples. These simple components should have few or, better still, no exceptions/limitations/restrictions because they cause the user to have to memorize them or may confuse the user. Furthermore, these simple components should be able to be combined, again in a simple, comprehensible fashion in order to build more complex, sophisticated and powerful components of the language which may be utilized by more experienced users, yet within the grasp of the novice who has learned the simple "building blocks." These more complex components should be developed by a consistent application of simple components and be given expressive names which clearly indicate their function.

By natural, I mean familiar, common, normal, or typical. The natural way of expressing something definitely varies from person to person, however if the programming language can emulate written and/or spoken English along with its syntax, then one should be able to maximize the number of people who would at least understand what you were telling the computer to do and thus learn how to do this themselves (provided English is understood in the first place). Terminology used by a Fourth Generation Language, then, should be natural and consist of sentences or phrases of English

words which are readily understood by a majority of the population.

By logical, I mean the language should incorporate only those functions and operations which follow from clear reasoning to avoid any "fuzzy" or "gray" areas. Given certain conditions or facts then only fixed conclusions could be reached. This will also reduce the requirement to remember these illogical areas and hence aid the user in learning the language since the user can be assured that all functions/components operate logically.

To learn all the details of a programming language, it normally takes many hours of practice and experimentation. To be easy to learn, the language should have few concepts/components/operations/functions that must be absolutely memorized. In addition, it must possess an easy,quick way to obtain answers to questions and determine options or courses of actions that the user has available to him from his present state. This latter requirement necessitates the use of an extensive help environment which dynamically tracks the user's position at any point in computer session.

B. EASY TO USE

For a programming language to be easy to use, it should permit someone with limited programming experience the opportunity to obtain meaningful results with a minimum amount of time spent designing, coding and debugging a specific application. In addition, the language should be flexible enough

for experienced users and with enough available shortcuts to be useful for them. The language should accomodate or support both commercial and scientific application program development, database creation and manipulation, report generation and graphics applications for relatively inexperienced computer users. Additionally, experienced computer users may also benefit from a language which allows easier development, debugging and maintenance of systems applications. A Fourth Generation Language will minimize the necessity to explain in detail how some application need be executed. Instead it should allow the user to just explain what he wants done and then let the computer, which utilizes an extremely versatile and flexible default mechanism, accomplish the task without the user getting involved in the "how" of the problem. However, the user should have the option to get involved with the "how" of any task through the manipulation or modification of the default structure behind the task(s) identified by the user. Minimizing what the user must know about how to implement or execute a particular function, operation or task, yet permitting him the opportunity if he wants it, will reduce some of the procedural details the user must remember and allow him to think more in terms of results rather than "how am I going to do this." A Fourth Generation Language will allow the user to just specify what actions he wants performed with no mention of method, order of execution or particular technique--a strictly non-procedural approach

to programming. This non-procedural aspect of a language has been demonstrated effectively by several database management systems but this should also be expanded beyond databases and into application program development as well. Current data manipulation languages (DML) are non-procedural in their execution of simple queries but for more complex queries, the DML is frequently embedded within some host language (e.g., Ingres within 'C'). However, a Fourth Generation Language will not require this embedding but will incorporate the DML and the host language into a single integrated language. This will facilitate the development of report and graphics generators as well as complex database applications since a user will only have to think about what he wants rather than how he will implant the correct DML queries within the host language. Thus, the Fourth Generation Language will be easier to use for database applications because the interface between the DML and the host language has been removed and the two languages integrated into one functional unit. Also this integration will reduce debugging time since an error prone interface has been removed. Maintenance of these languages will be simplified because they will be easier to understand due to fewer details explaining how the interface was performed or how a particular task was executed. So not only will these Fourth Generation Language programs be easier to write but maintenance costs should be reduced since understanding what a program does, reusing parts of

it and modifying other parts of it should be relatively easy and not complicated by excessive, procedural or interfacing details.

For ease of use, a menu driven system works well, especially for users with limited experience. The menu contains a listing of options or courses of action open to the user at any stage of program development. This listing alleviates the necessity of memorizing commands or sequence of commands since the user only has to refer to the menu and then select an appropriate action. Besides, "recognition is easier than recall." (Helander, 1981, p. 304) The use of menus also restricts the user to only legal commands, thus reducing the time spent pursuing illegal commands or fallacious courses of action. However, the option to allow the user to specify what to do with command sentences or phrases should not be overlooked, especially when dealing with experienced users. Menu-driven systems tend to be less flexible and sometimes burdensome to use if one has to go through several menus to perform a task that may take only one command phrase to execute in a command-driven system. The flexibility and usefulness of menu-driven systems are a function of how well they have been designed to handle the variety of tasks and methods of operations demanded by ingenious users. Optimally, a Fourth Generation Language should be both menu-driven for ease of use but allow for custom designing of menus or be command-driven for the flexibility required by experienced users.

Simple, flexible control structures which are capable of doing all the operations/functions which a Third Generation Language can do are also necessary. The next generation must be at least as powerful as its predecessors. Regularity should be adhered to in the design of these control structures so that no special cases need be remembered and the format of each control structure follows similar templates.

An extensive help mechanism is particularly important in making a language easy to use and it should be readily available through an online approach. This help mechanism should be attuned to a user's current condition or state and offer the user options which are applicable and meaningful to the current situation. A call for help should not result in a flood of superfluous information which would only tend to confuse the user. For this reason, the help mechanism must track where the user is at any point in time, i.e., what commands he has issued, what actions have occurred and what is his current status. An options table should be made available to the user which is based upon his current situation and how he got there. Only viable options would appear in the options table. The user should be able to return to hiis previous state or to a main menu or to the operating system upon request. Once help is requested, brief statements highlighting alternatives will be displayed but if the user would like a more detailed explanation, each help option should also be backed up by a more thorough description of that particular option--its meaning, cause and

effect. The help mechanism should employ windows which do not entirely rewrite the screen each time help is called. Instead, as much of the screen should be preserved in order to allow the user to read both the help comment plus the point from which he requested help.

C. POWERFUL

A Fourth Generation Language must also be a powerful language, retaining and/or enhancing the capabilities of the previous generation languages. A powerful language must provide operations or be functional in many areas and handle not only simple computing problems but complex ones as well. Essential elements include applications development in both the commercial and scientific arenas, report generation, a graphics package, communications with other computers and database creation, access and maintenance. Hence, a powerful language is fully functional or useful not only for the new computer user but the computer professional as well. I equate powerful with productive and effective use of the computer by the wide range of potential users.

Although it would be preferable for all application development to be non-procedural, this is unrealistic and at this juncture incompatible with the conceptual logic utilized in problem solving. Thus, a Fourth Generation Language must incorporate the necessary, procedural control structures of the previous generations in order to perform

the conditional, iterative and control flow functions needed in application development. Conditional statements such as:

(1) If-then

(2) If-then-else

(3) Select or case

are necessary for testing if certain named conditions exist and indicating what action to take when they do exist. At least three types of iteration are important to retain:

(1) Loop quitloop endloop

(2) Loop [numeral]/[variable] endloop

(3) Loop skip-one-iteration endloop

The first iteration type permits both iteration until a condition becomes true or while a condition is true when the quitloop part is executed after a test condition is checked. If the quitloop check is performed late in the loop, after all other statements, you have the repeat-until construct. And if placed before other statements, you have the while-do construct of third generation languages. Also, this first type of iteration permits exiting a loop anywhere within the loop and not just at the beginning or end of the loop construct--different from most third generation languages. The second iteration type allows looping a fixed number of times as indicated by the numeral or variable upon entering the loop. This is like the for-do construct of previous generations. The third iteration type allows one to skip a single iteration of a loop but 1.) return to the beginning of the

loop to continue again until a quitloop condition is reached if used in conjunction with type one loops or 2.) return to the beginning of a loop, advance the count and continue looping until the count is zero if used with type two loops. So, the skip-one-iteration does exactly as its name indicates and must be used in conjunction with one of the other two loop constructs, but it has no counterpart in previous generations, apart from the use of GOTO's. The skip-one-iteration portion of a loop construct may be used with or without a conditional statement inside the loop. The purpose of this skip-one-iteration is simply to add greater flexibility to loop constructs by allowing the user to easily skip one or more iterations of a loop without having to exit the loop altogether.

To aid the user in utilizing the proper format of the above constructs, an interactive environment which checks the syntax of the user's program as he types it in, must be a part of the Fourth Generation Language. The system indicates immediately if the conditional or iterative construct is not formed correctly or a reserved word is misspelled. It also rewrites the construct using identation to make the reading of the finished program easier. Another approach to ensure error-free use of these constructs is to allow the user to select the construct he wants to use from a list of possibilities displayed in a window on some portion of the screen. Then once he has made his selection, the correct

format is automatically reproduced on the screen and he need only fill in the blanks. And if constructs were nested within each other, the format would also be automatically indented to reflect this nesting, thus enhancing program readability. If this list of options is used in conjunction with an interactive environment, then the system would only need to check the syntax of the conditions and statements the user has typed into the blank spaces of the construct.

A Fourth Generation Language should permit the user to name and define functions or procedures which he may then use in his current program or in future applications. 'Function/procedure' would be listed in a window along with the above mentioned constructs and once the user selects 'function/procedure' from the list of options in the window, the system prompts him for the name, parameters and body of the function/procedure. Upon completion of the function/procedure, the name of it is added to either a separate window or to the constructs' window, to permit the user to use it again by simply naming it and providing any necessary parameters. Each time a function/procedure is used again in the program, the system would check the parameter list to ensure that it was correct in number and type of arguments and warn the user if not. Each function/procedure should also be able to be executed separately to ensure they are error-free. In fact, all fragments of a program should be capable of being executed independently and before the whole program is done, in order to reduce time spent debugging.

Data access and maintenance authorized to the user should also be possible in the application environment of the Fourth Generation Language. The following database management functions should be allowed (again, where authorized): retrieval, deletion, insertion, modification and the use/development of aggregate functions. Any joins needed in the execution of any one of these database management functions should also be permissable but will be entirely invisible to the user. The database management tasks, like the control constructs must also be listed in a window. The user then picks which one he wants and the format of that particular task appears on the screen and the user need only to fill in the blanks. The user could also query the system about the database organization to see the makeup of the internal relations, i.e., table names, field names and aliases, indices and even an entity-relationship diagram of the schema. The user must be able to ask the database management system which relations contain a specific field or set of fields, as well as establish his own aliases. However, the most important point here is that the user does not have to change environments when working with the database or when working with applications programs. Both have been integrated into the same environment and the syntactical requirements (if any) for both are identical. From this single environment, the database can be created, have user views defined, establish authorizations, initiate integrity checks as well as perform

the above mentioned functions. There should be no explicit context switch when changing between tasks or programming tools such as switching from editing, to program execution, to debugging, etc. In addition, the various ongoing tasks could be displayed in different windows and the user could then manipulate these windows and/or their contents, simultaneously or one at a time. This would permit the user to define programs, edit them, ask for system help or send and receive messages via different windows.

Another powerful and extremely useful tool or command is the cancel command. This allows a user to backtrack or cancel previous commands and get back to a point he chooses to proceed from again. And if he did not really want to backtrack at all or so far he should be able to cancel the cancel and effectively go forward again.

The burden of declaring variable types must be lifted from the user of a Fourth Generation Language, but left as an option that experienced users may choose for efficiency or accurate documentation purposes. A Fourth Generation Language should be capable of assigning appropriate types itself from the user provided data as well as remain as flexible as possible when operating on variables of dissimilar types--such as real/integer integer/real, string/char, char/string, etc. All these operations should be equally possible without generating error messages. To be at least as flexible or powerful as previous generations of languages,

a Fourth Generation Language must encompass all the data types utilized by these earlier languages, as well as allow the same or more kinds of operations on these data types.

IV. DESIGN OF A FOURTH GENERATION LANGUAGE

Although Third Generation Languages have emphasized a textual approach to making computing languages powerful, as well as easy to learn and use, text by itself has proven to be rather restrictive and one dimensional. Currently, there are several graphical approaches proposed or already developed for use with databases and/or program development that seem to meet most of the criteria for a Fourth Generation Language (i.e., APPLICATION FACTORY, NATURAL, QUERY-BY-EXAMPLE, etc.) but these languages lack the explanatory information one sometimes needs to understand a technique or procedure. However, a textual approach fully supported by graphics may be a fair compromise between the two approaches mentioned above since it borrows from the strengths of each, i.e., graphics being universally understood, powerful in conceptual depiction and easy to learn and use (Ives, 1982, pp. 21-23); text is rich in semantics, expressive and possibly a more concise and less awkward manner in which to develop a simple application or query and an effective way to call for a report, program or subroutine to be executed. Thus, a graphical-textual modus operandi enhanced by menus and versatile window or viewing methods may achieve the goals desired of a Fourth Generation Language, while alleviating the weaknesses of using purely textual or graphical methods.

In order to make programming languages easier to learn and use, more of the abstract properties of programs should be brought out on the computer screen. What the user sees on the screen should be related closely to the user's own problem solving thought processes, as well as his own physical perspective of the application/system he is trying to create. Programming languages consisting of text only, do not sufficiently reduce the abstract qualities of programs to easily understood and easily visualized segments whereas graphical/pictorial methods of representing programs can be utilized to simplify and/or categorize abstract portions of programs. How much of an application program should be textually or graphically represented is a moot point, but because of the reasons listed below, I believe graphical program representation will be favored over textual methods alone and some combination of the two representations, with an emphasis on graphics, will prove particularly vital in helping Fourth Generation Languages achieve their goals.

Since the human mind is strongly visually oriented and since people acquire information at a significantly higher rate by discovering graphical relationships in complex pictures rather than by reading text, the importance of graphical program representation seems obvious. Pictures are much richer, more powerful vehicles of expression than a one dimensional stream of text. The properties that pictures borrow from the real world include shape, size, color,

direction, distance, multiple dimensions, and texture. Using these properties in the encoding of information can result in more compact program expression and a resultant decrease in decoding time. Our eyes provide instant, random access to any part of a picture and to detailed and overall views as well. In contrast, access to text is strictly sequential. The use of objects from the real world in pictures that are used to illustrate abstract ideas makes the ideas simple to think about. Good pictures that incorporate real world objects and are used to depict abstractions will help not only the specialist to formulate and communicate thoughts faster and better, but will also aid the novice who is groping for a handle on unfamiliar matter. For example, the best pictorial aid for learning computer programming is a concrete model of the computer. Using text alone forces the user to come up with his own mental images. It seems then that graphical/pictorial program representation is essential for making programming easier. Since graphics offer advantages over text alone, the use of graphics should be extremely useful in depicting the three major components of programs--control flow, data flow and data structure (static as well as dynamic). (Raeder, 1985, pp. 12-15)

A simplistic but I believe true view of computing in general, is that computing/data processing is always a transformation and/or transaction process such that a user inputs some data/information/request and expects an answer--some

new data/information/yes-or-no/report or simply that no answer is available. Continuing with this view then, computing can be considered to be composed of inputs, outputs and the transformation/transaction that created the output from the input. Speaking strictly from an end user (and possibly the application programmer) viewpoint, it would be beneficial if he could simply define his input, define the desired output and in the highest non-procedural (procedural only as a last choice) language possible describe what must be done to the input to obtain the output. It must be noted here that since problem-solving itself is a procedural process, the total elimination of looking at a problem from a step at a time approach seems unlikely. However, within each problem-solving step there may be numerous approaches, implementations or design techniques for handling the problem and here is where the end user may benefit from the non-procedural methodology of just telling what must be done and letting the computer accomplish it in the most efficient manner possible.

Since either an entirely textual or totally graphical approach seems to be unacceptable, the entire system of input-transformation/processing-output should be automated and depicted as a combination of text and graphical structures (pictures). So, just as hardware development has become increasingly automated through various CAD/CAM techniques, Fourth Generation Languages can take the next step forward in software automation.

A. DESIGN OVERVIEW

A specific idea may be to just automate the current structured analysis and design techniques through the use of a friendly user interface involving a textual-graphical approach along with "pop-up" menus, windows, an interactive environment and other potentially user friendly innovations. "Pop-up" menus are simply listings of actions or commands that are legal within the context of the window they "pop-up" in. As a user progresses in his development of a program, different actions may become legal and thus, a new menu pops-up in the window. A window is a bounded, physical location on the computer terminal screen, in which the user performs the various tasks needed to develop, test, execute and maintain a program. Multiple windows can exist at the same time on the screen, but only one task at a time can be performed in a window.

End user applications could be depicted as hierarchies of data flow diagrams which would in turn make these applications more understandable since they "are a natural way for people to depict applications." (Stevens, 1982, p. 172) This would permit end users an organized, yet simple approach to problem solving and more experienced users the opportunity to perform analysis, design and prototyping with initial and future specifications in an automated manner. Since each phase of development could easily be saved and some self-documenting items could be added to save the programmer from

some of the more tedious but necessary features of software development, documentation could be improved and maintenance costs reduced. Higher Order Software (HOS) already has a graphical method of displaying inputs being modified into outputs, however I beleive this could be elaborated in order to provide end users an easier system to use and experienced users a more powerful system. It could also provide for the integration of database applications (Martin, 1985, pp. 252-257).

Initially, when a user first calls upon the textual-graphical system a menu should be made available to determine the user's purpose, e.g., run a program, develop a program, query a database. Or more simply, the user could be prompted for his input and from this the system would then know the user's purpose. If the user is running a program, he may need the system to remind him of the program's name and how to cause the system to execute it. The input here could be the program name and the output would be the execution of the program. If the user wants to query a database, one of the current graphical interfaces (QBF, GLAD, etc.) could be used to walk the user through the development of his query. The input here could be the database name or the query itself and the output the answer to the query or the introduction to the graphical interface. However, if the user wants to develop an application program (and any application will handle database accesses as well) the system will walk

him through its development from the highest level or layer down to the lowest necessary for actual computer generated code. Input here could be the highest level of incoming data and the output could be the data, report, or answer the user is looking for.

For a simple application, the user needs only to provide the name(s) of the data input and the system could ask the user the data types the user has in mind for these inputs by providing him a list of possible types. The same for the output would occur and the system would also display the input(s) as named arrowheads entering a window (the location of the data transformation process) and the output(s) would be automatically displayed as arrowheads exiting the process window as indicated in Figure 1.

Figure 1. Simple Application

Next the user is prompted by the system for the contents of the process window or a description of the action performed within the window. For a more complicated application, the system asks for intermediate input(s) and output(s) and provides the additional data transformation windows. For the simple, one layer program, the user would be aided in this development of the contents of the process window through on

screen or "pop-up" menus displaying the possible commands and/or control structures available to him. If control structures were utilized, "action diagrams" (Martin, 1985, pp. 157-172) would be employed to ensure the proper format was followed. Action diagrams are simple diagrams which reflect the structure of a construct through the use of brackets, boxes or other graphical pictures. They are particularly useful in the design and understanding of complex code. A user of such diagrams can tell quickly whether a construct is structured correctly and completely or not. If database access was needed, then GLAD (Wu, 1985, pp. 12-20) with its built-in program box could be utilized to express the entire data transformation or a portion of that action. And the database access would be depicted as named arrowheads to or from a named database as shown in Figure 2. Anytime the

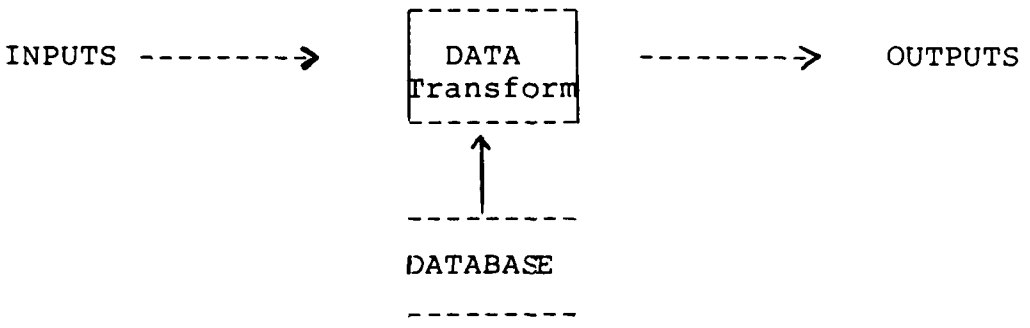


Figure 2. Database Integration

user tried to perform some inappropriate operation on the input or output data the system would caution the user. Upon completion of the actions within the process window the

user could select to execute the module to ensure proper functioning. The system would prompt the user for any documentation deemed essential to the understanding of the module and this would be stored separately and only be displayed if requested. In the more complex applications, as the user decides on the appropriate intermediate input(s) and output(s), the system provides the necessary process windows and the user must name them or describe their contents as he would do in a simple application. The user continues to break down all the input and output to their lowest levels, or until he can think of no more intermediate data, then he proceeds to complete the contents descriptions of any windows not already described. What the user will end up with will be a hierarchy of data flow diagrams from the most coarse view to the finest view, where actual code exists. The system keeps track of each level in this hierarchy and at any point in the development of the application the user may select to 'zoom-in' or 'zoom-out' one or several levels to review, modify or execute what he has already described.

The user may also elect to have different portions of his program in view by specifying window locations and the contents of these windows and he could be performing different tasks in the different windows, such as, documenting a module, describing the contents of a process window, executing or possible debugging a process window or entire program. The action of each window would be kept as the status or

state of the window and the user would be allowed to move freely between windows, call upon menus that were applicable within the various windows, close or open windows and adjust the size and print of windows. The use of windows is quite natural and supports people's workhabits since when most people perform a task, they tend to shift their attention from one subtask to another in a fairly random fashion. In addition, the use of windows could alleviate the awkwardness one experiences when going from one part of a system to another (changing modes, i.e., from editor to operating system to debugger). With multiple windows, the user can switch the mode he is in by simply moving to or opening another window and still keep his other activities in sight if needed (Raeder, 1985, p. 17).

B. COMPONENTS

The data flow view of an application is a natural way for end users to depict a solution to their problem. Typically, people perform functions and share data with other people who have other functions that they perform. Data flow diagrams picture these interrelationships and flows of data between independent functions as they exist naturally (Stevens, 1982, p. 172). The components of this approach to application program development are inputs and outputs, processes (functions or data transformations), databases (files), function libraries, menus, and windows with their associated

commands, herein called edge commands. A discussion of each component as it relates to a user-computer session follows.

When a user begins a dialogue or session with the computer where this textual-graphical approach has been installed, the user may be presented with a main menu to select his initial activity--such as, program development (and this is integrated with access to a database, if necessary), program execution or database activity. This main menu may not be necessary if each action taken by the user can be classified as either an input, process description or output. For instance, a database query could be broken into these three parts where the database name is the input (user could deal with multiple databases), the development of the query itself by means of some user friendly tool is the process description and the result of the query is obviously the output. If the user decided to create a database, the input would be the user selected name for the database. The system would check the name for uniqueness and then, if there was no main menu to indicate that this was going to be a database activity, the computer would ask the user if this name was for a program, database, file or function library. The user would select database and then the system would guide the user through the development of the mandatory and optional features of his database and then prompt him for values to enter into his database. All of this would be considered the process description. The output would consist

of a list of options for the user where he could designate the output he desired--hard copy of database, print database to screen or maybe nothing at all. When the user completes the input, the process description and the designation of the output he would select a command on the edge of the screen labelled 'end'. If the user did not complete development of his database or some other activity then he could select an edge (edge of screen) command such as 'quit' or 'cease for now' from a set of commands always present along the edge of the screen. If 'quit' or 'cease for now' was selected, he would be prompted if he wanted to save what he had done so far. Later, upon re-entering this activity he could be prompted to see if he was continuing where he left off, changing part or all of what he had already done, starting anew or reviewing what he had done. In each case, the system would direct him to the correct part of the activity for further development. If the user wanted to update a database, the input would be the name of the database. The system would check if the name was unique and if not then it would ask the user if he was working with a database, program, etc. To distinguish between querying, creating, updating or deleting the named database, the system would ask the user which activity he wanted to do. The system must verify that the user has authority for the activity selected before proceeding. In addition, if a user is only allowed access to a particular view of the database, the

system must ensure that only this view is provided to the user. The system would guide the user to that part of the database he wanted to update by asking him the table name. Then "pop-up" edge commands such as 'table insert', 'table delete', 'record insert', 'record delete', 'field modify', 'field delete' could be selected by the user to match what he wanted to do and each of these commands would guide the user to the proper location within the table. Once again the user could select 'quit' or 'cease for now' if he had to stop before completing his actions or he could select 'end' if the update was done. The output would be a list of options for the user just as in the database creation activity.

All of the above activities which have dealt with a database alone would be performed through the use of menus, edge commands, system prompts and multiple windows. This requires little or no use of pictures unless the edge commands used suggestive icons to indicate the command's functions.

C. APPLICATION DEVELOPMENT

For application program development, the system aids the user by organizing the development process in a top-down fashion. The final program consists of from one to many levels of data flow diagrams where each level is a refinement of the one above it. When beginning a session with the computer, the user would indicate the activity of application program development either by selecting it from a menu or by indicating the name of the program and the system

would then ask him which activity he wanted. Once program development had been selected, the system would add the commands in Figure 3 to the edge of the screen. From this

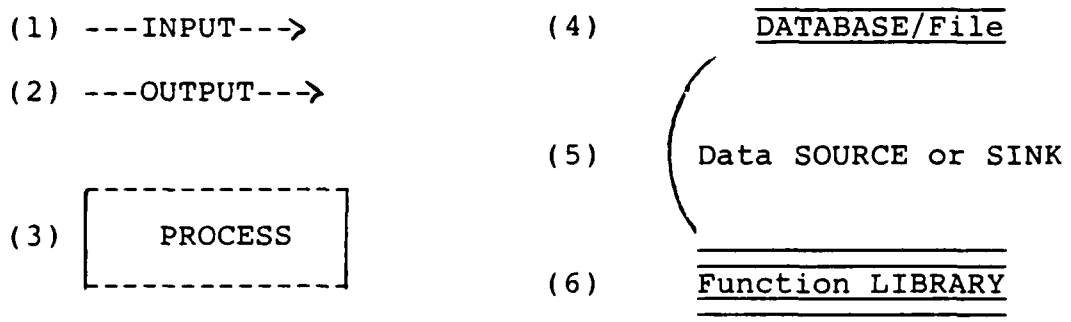


Figure 3. Edge Commands for Application Development

point, the user would be expected to either begin his program development by selecting one of the new commands indicated above or select help for instructions on how to proceed. In one corner of the screen, the word UNDEFINED would be displayed if any portion of the program being developed was not complete. In addition, the user could select the word UNDEFINED to obtain a listing of the incomplete parts of the program.

If the user selected the input arrow, a window could be opened in the center of the screen and it would contain the information denoted in Figure 4. The user would then be expected to fill out this form about the input data as best he could, but fully realizing that this information could be changed as the program developed. Data that was identical but that had different names could all be listed in a single window by placing the different names after the first prompt. For data with different sources, types or range of values,

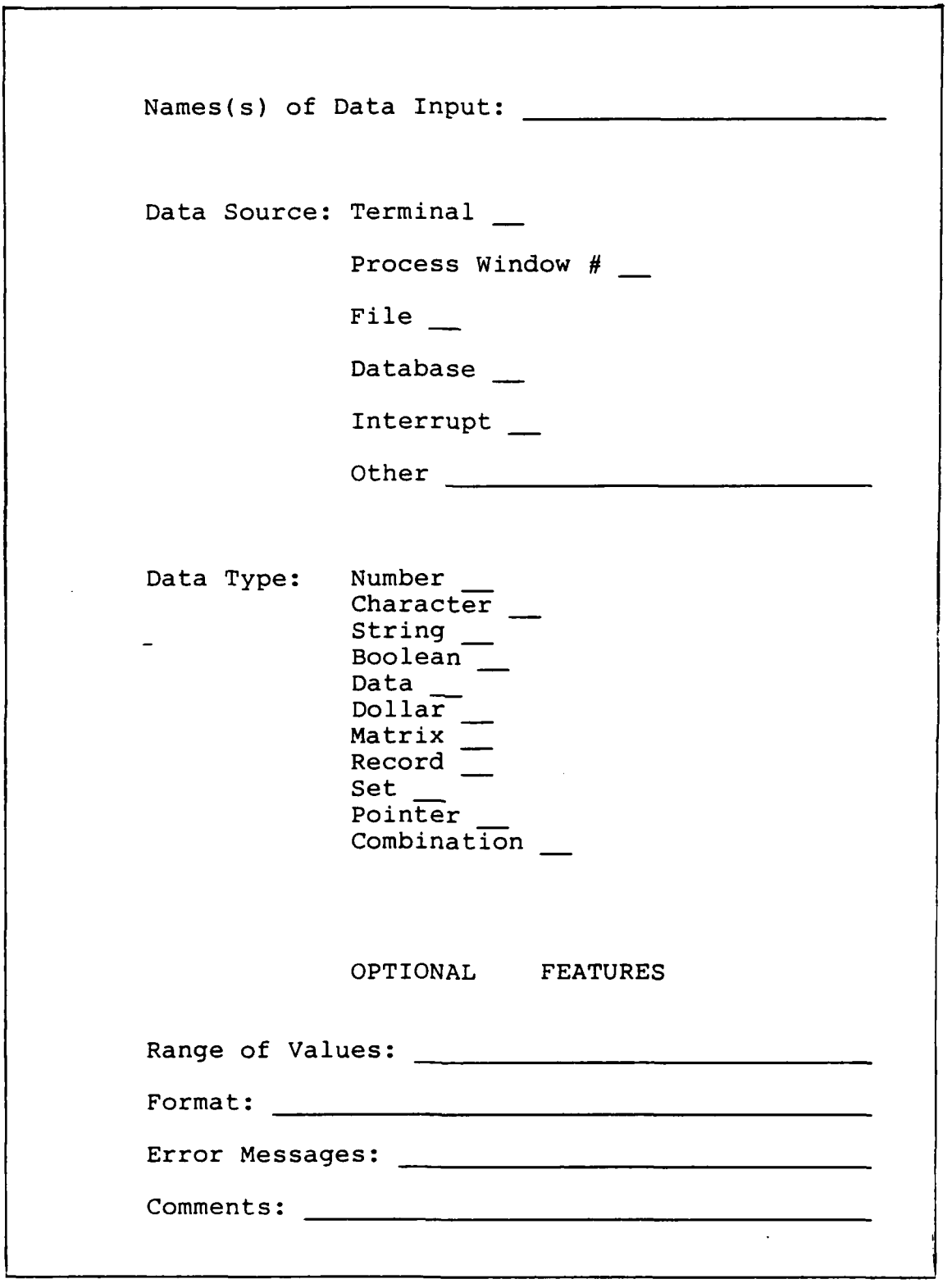


Figure 4. Input Window

separate input windows for each would be used. Under the data source category, if the user indicated that the input was coming from a file or database, then the system would prompt him for the name of the file or database. In addition, the system would use the information about the source to determine the type of I/O to use in bringing the input into a process window or sending as output.

Under the data type category, if the user selected a matrix, record or set type of data structure, the system would prompt him for type of data structure(s) they contained. The system would also ask for the size of the matrix. If the user selected the combination data type, the system would prompt him for which other data types were combined and in what order or format.

The bottom portion of the input or output window would be for optional information which the user would not have to complete before he finished development of the application, however all other information would be mandatory and thus have to be completed prior to completing the development, otherwise the system would warn the user of the incomplete input or output window. The format section would be used if the input came from a form or the output was going to be in a particular format, then the system would guide the user through the design of the form. Input/output error messages could be established here so that, if any of the features of the data description were violated, an error

message would be issued and the user allowed to begin again. Comments are just user documentation.

All this information about the data used by the program would be consolidated in a data dictionary which the user could look at upon request. Cross-referencing information such as where data was used, where it came from and where it went would be automatically generated and kept in the data dictionary.

The window used for output would be identical to that used for input except that data source would be changed to data destination. And as with all windows within this system, the user would be free to move the input/output window, enlarge, shrink, open, close or cover and uncover it. The main difference between closing and covering a window is that closing means you are done with the window while covering means to hold onto the contents of the window just as you left it, but hide it from view until the user asks to uncover the window.

If the user selected the process box and he was just beginning development on a program, then the user would see a window with a number 0.0 located at the top of it appear in the center of the screen. This window is depicted in Figure 5. The user would be expected to indicate all input(s) and output(s) to this high-level view of his program and the system would check to see if these input(s) and output(s) were defined yet. If not, the UNDEFINED indicator in the outer

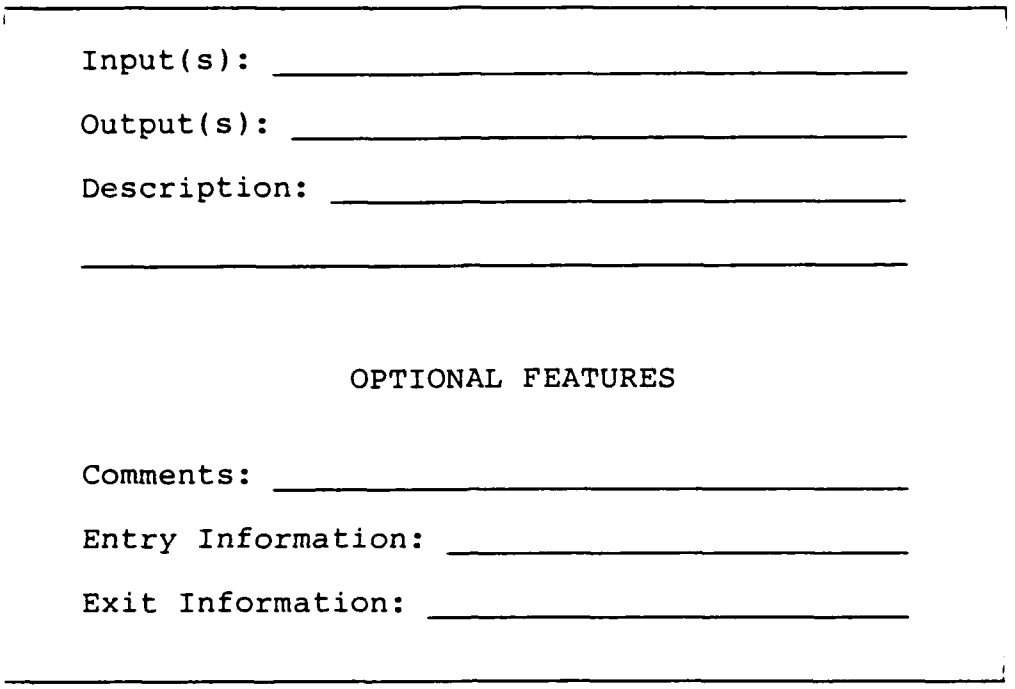


Figure 5. Process Window

window would "light up." As the user indicates the input(s) and output(s), named arrows would be shown entering and exiting the window and the type and source of the data (if known) would be depicted by icons attached to the arrows.

Next, the user would have to describe the process(es) he has in mind for this particular program. But since the 0.0 level is the very top-level, this description would normally be rather general and usually no code would be written here. The comments section is again just user documentation. Entry and exit information are to be utilized to handle control flow within the system. Any valid statement that can be placed in the description of the process window can also be placed here. The effect of the entry and exit information is to control entry and exit to subordinate levels of

process windows--the system must check the entry information of the higher level window before entering the lower level process window and the system must check the exit information of the higher level before leaving a lower level. This provides the necessary control flow needed in the development of an application.

Upon completing the process window, the user would select the 'end' edge command and this window's information would be saved and the screen would change back to the original program development one. Then, each subsequent selection of the process box would cause the system to prompt the user for how many process windows he wants. After the user entered a number, the system would provide the number of windows asked for and each would be automatically numbered in accordance with the appropriate level the user was working at. And if too many or too few windows were requested, the user could simply close or open additional windows and again the system would automatically number the windows, if necessary. If too many windows are present, the user can always move them or cover temporarily the ones not immediately needed and enlarge or shrink the windows to the desired level, which automatically adjusts the size of the lettering and/or pictures in the window.

Just as at the top-level, the user needs to describe the input(s) and output(s) to each proces window and then describe the process which transforms the input data into

output data. To assist in this area, the system provides the following additional edge commands for the user to use in developing his program:

(1) If-then

(2) Loop

(3) Function library

(4) Database access

When the user selects the if-then command for use within a process window, a window, like Figure 6, appears where he was working. The selection of the if-then command provides

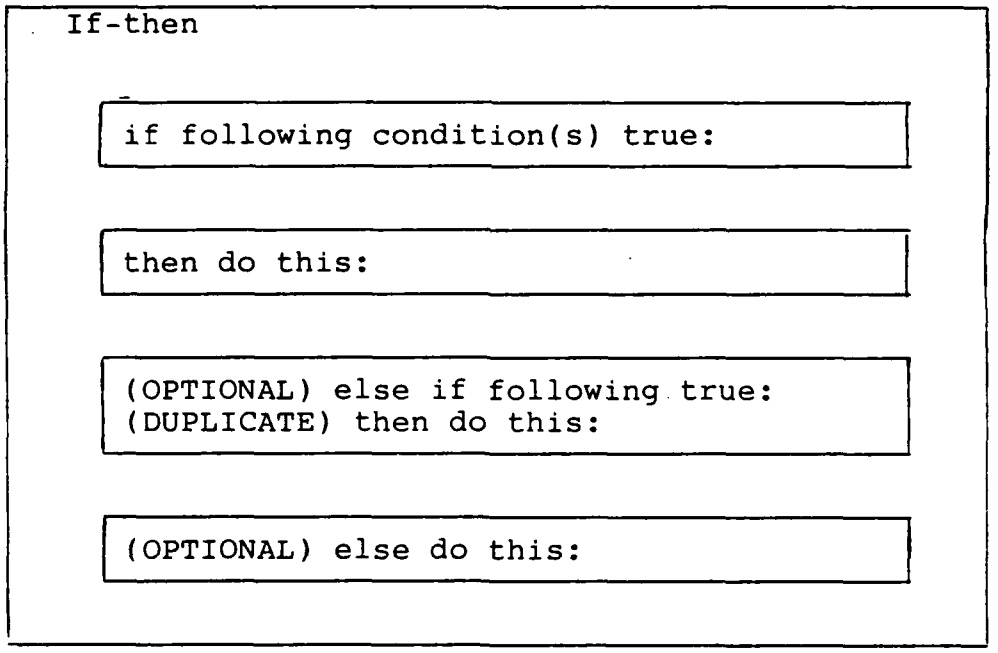


Figure 6. If-then Construct

the user with the following possible constructs: if-then, if-then-else, if-then-elseif, and if-then-elseif-else. The user only needs to type in the condition(s) he wants checked

and the resulting action(s) to occur if the condition(s) is (are) true for a simple if-then statement. The user is allowed to use any statement in the condition section of the if-then construct that evaluates to a true or false. If the user enters an incorrect statement (one which does not evaluate to either true or false), the system highlights the incorrect statement and if the user asks for help, an error message is printed explaining the problem. The user may use the following logical operators with his list of conditions: and, or, not. If the user has alternative condition(s) or alternative action(s) then he simply fills in these optional portions of the construct. If he has more than one alternative set of conditions and actions, then he can duplicate the else-if part as many times as necessary. As the user fills in each part of the construct, the enclosed parts of the construct expand automatically to accomodate the condition(s) and action(s). The system permits nesting of these constructs to any level but will not remove the enclosing boxes for each part of the construct until the entire construct is completed. Then, the system places the if-then construct in the correct location of the developing program and with proper indentation reflecting the levels of nesting. The purpose of handling the construct in this manner is to ensure the proper usage of the construct and to help the user understand the needed parts of the construct, thus making it easier for him to use by avoiding common syntax errors.

When the loop construct is selected, the user is presented with a similar situation as with the if-then construct. With the loop, he would see a window, where he was working, like Figure 7. Depending on where the exit condition is placed,

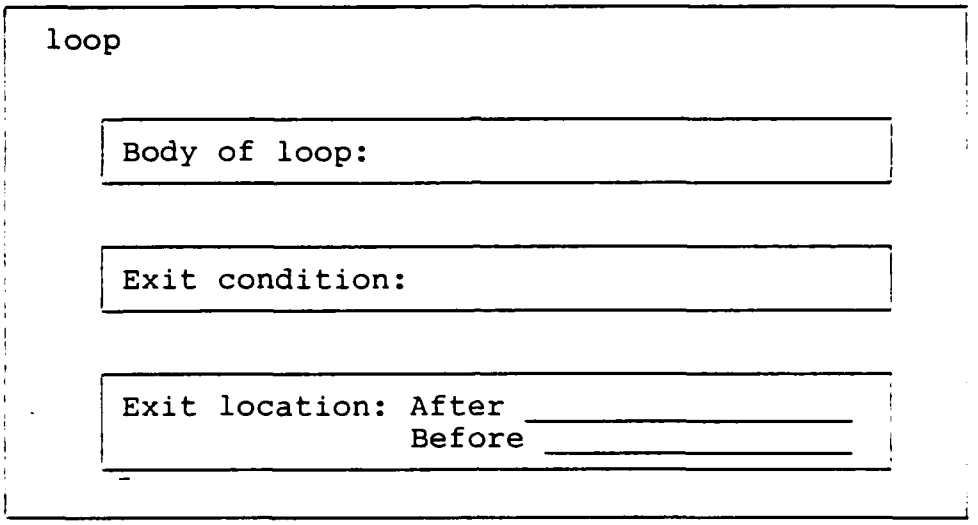


Figure 7. Loop Construct

the user can have either a repeat-until or a while-do construct commonly found in Third Generation Languages. Additionally, the user could position the exit condition in the middle of the body and thus loop through part of the body more times than the other part of the body--a mid-loop exit. The body of the loop would be the statement(s) that the user wants to have repeated and this could also contain other loops nested to any level desired. The exit condition must be a statement(s) that evaluates to either true or false. The location allows the user to describe precisely where he wants to place the exit condition for evaluation within the

body of the loop. He could place the exit condition either after, before or between statements. Once again, after the loop construct is completed, the boxes are removed and the loop construct is properly indented and positioned next to the other statements already in the developing program. Iterations free of syntax errors are thus guaranteed, and user understanding as well as ease of use is enhanced.

If the user needs to correct or edit some portion of the program already written he simply moves to that segment and uses the 'insert', 'delete' or 'typeover' commands to make correction. Once the user attempts to make a correction to either the conditional or iterative constructs--the system provided boxes, used to initially build these constructs, reappear and once again help ensure correctness and completeness of the construct.

During development of a program, the user may develop functions that he wants to save for future use or he may have a need for functions and procedures that he (or others) has (have) developed previously. Hence, the user selects the 'library' edge command in either of these cases. When 'library' is selected, the user is given a choice of adding to the library, deleting items from his view of it or listing its contents. If adding to the library, the user selects this option and the system prompts him for the name, the input and output parameters and the process window that contains the function to be added to the library. If the user

needs to use a function/procedure from the library, he could simple name it while in the process window or could ask for the listing of all library functions and then select one from the listing. Once he has selected a function/procedure for a window, the system checks to ensure the input and output parameters for the process window match the type for the function and if they do not match, the user is warned of the problem by an error message. The system will also have some built-in functions such as: sort, max, min, average, member and initialize. Utilization of such functions allows the user to specify just what he wants done and not how to do it, thus emphasizing the non-procedural approach to programming.

If during the course of program development the user finds that he needs to access a database for information which he will use within his program, then he will select the 'database access' edge command. The system will ask him the name of the database and once he provides the name, a window will open for him and within this window he will develop his query utilizing a non-procedural, graphical database language. Once his query is complete, he will select the edge command 'end' and the query will be automatically incorporated into the program.

While developing a program, if the number of process windows becomes too much to be easily seen on the screen at one time, then the system will "wrap" the process windows around to both the left and right sides of the screen.

Alternatively, the user may select to cover the process windows he does not need, which provides more room on the screen, and then uncover them when he is ready to work on them. If the user needs to shift the process windows either right and left or up and down, movement arrows will be located along the edge of the screen. If the user wants to visit other levels of development, then the two edge commands 'zoom-in' and 'zoom-out' would be used.

Once a user completes a process window and supposing the process is not to be broken down any further, he would be able to test it to ensure it functions properly, i.e., that the input is actually transformed into the output. The user would simply select the edge command 'test' and the system would prompt him for which process window and what input values he wanted to use. The system would then display the output and let the user know if it matched the output he declared. Module testing during development would reduce the degree of debugging necessary for each application.

D. EXAMPLE

Utilizing the textual-graphical, Fourth Generation Language described above for the development of an application program may help clarify some of the features of this language. Upon entering the system, the user selects the option to develop an application program. The "pop-up" menu which includes the commands--'input', 'output', 'process' and 'library'--appears on the edge of the screen or development window.

If the user first wants to indicate the top level description of his program, he selects 'process' to begin (Figure 8).

0.0

Input: fraction

Output: reduced_fraction

Description: Reduction of any nonzero, positive fraction to its lowest term.

OPTIONAL FEATURES

Comments: This program accepts any positive, nonzero fraction and reduces it to lowest terms. The user is prompted for a fraction, the program determines the greatest common divisor (GCD) of the numerator and denominator then uses the GCD to divide both the numerator and denominator to obtain the reduced fraction.

Entry Information: Print "Enter fraction or 'q' to quit. Example - 6/12."

Exit Information: Terminal input = 'q'.

Figure 8. Sample Top-level Process Window

After filling in the above information, the user chooses to describe the input at this top level. This is completed by the user as indicated in Figure 9. From the input window,

---

Name(s) of Data Input: fraction

Data Source: Terminal

Data Type: Combination - Number Character Number

OPTIONAL FEATURES

Range of Values: Number = 1 - maxint

Character = '/'

Format: numerator/denominator

Error Messages: Violate Data Type - "You must enter a number followed by a slash followed by another number with no intervening spaces. Please try again."

Violate Range of Values - "Enter only positive numbers between one and maxinteger. Use only a slash between the numerator and denominator numbers."

Comments:

Figure 9. Sample Input Window

the system can tell the name of the data, where it will come from and any constraints imposed on the data by the user. Next, the user decides to fill in the output information for

Next, the user decides to fill in the output information for the top level as shown in Figure 10. At this point, if the

```
Name(s) of Data Output: reduced_fraction
Data Sink: Terminal
Data Type: same as input
                    OPTIONAL FEATURES
Range of Values: same as input
Format: same as input
Error Messages:
Comments:
```

Figure 10. Sample Output Window

user asks for a display of his program so far, the system would display the following:

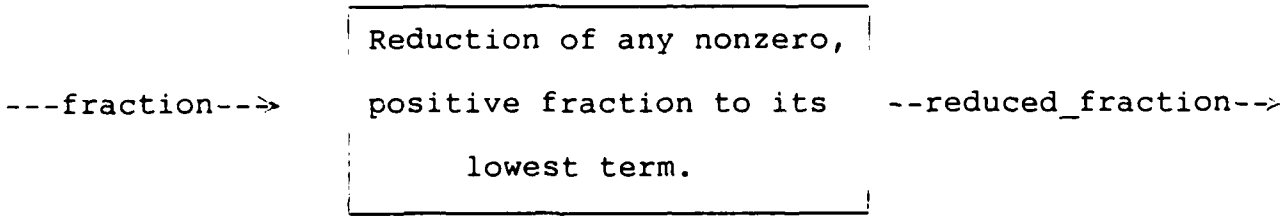


Figure 11. Sample Program Description

Now the user must think about how many other process windows he will need for the second level of the program. However, if he over estimates or under estimates, he can always use edge commands to delete or create more process windows. Assume our user has estimated accurately and has asked for

two more process windows. Both windows will appear on the screen and he can choose either to begin filling in or he can cover one and enlarge the other to see what he is doing a little better. Our user elects to work on one at a time and thus covers one of them by using the 'cover' command. His work is shown in Figure 12. Note that the variables

---

1.0

Input(s): fraction

Output(s): fraction, gcd

Description:
MAX(numerator, denominator) -> large
MIN(numerator, denominator) -> small

LOOP
Body of loop:
1. large MOD small -> remainder
2. small --> large
3. remainder -> small

Exit condition:
small = 0

Exit location:
Before 1.

large -> gcd

OPTIONAL FEATURES

Comments:

Entry Information:

Exit Information:

Figure 12. Sample Level One Process Window

"numerator" and "denominator" have already been described implicitly in the definition of the input data named "fraction." Also, temporary variables "large," "small" and "remainder" must be of the same type as the variable(s) that assign (designated by the arrow -->) to it values. The loop construct I left as it would initially appear to the user. Normally, upon completion of the loop construct, the system would automatically convert it to code. The above process window is a complete module in itself and could be tested by the user by selecting the edge command 'test.' However, the user has created some intermediate data which must be defined. He decides now to define this data called "gcd," instead of proceeding to the covered process window. The data is defined in Figure 13.

Name(s) of Data Input: gcd

Data Source: Process Window # 1.0 to # 2.0

Data Type: Number

OPTIONAL FEATURES

Range of Values: 1 - maxint

Format:

Error Messages:

Comments:

Figure 13. Sample Intermediate Input Window

The above data description informs the system that "gcd" is passed from process window #1.0 to process window #2.0 and that it is of type "number."

Finally, the user is ready to use the 'uncover' command to complete process window #2.0. Results are shown in Figure 14. Upon completion of process window #2.0, the user

---

2.0

Input(s): fraction, gcd

Output(s): reduced_fraction

Description:

```
numerator ÷ gcd --> new_numerator
denominator ÷ gcd --> new_denominator

new_numerator/new_denominator --> reduced_fraction
```

OPTIONAL FEATURES

Comments:

Entry Information:

Exit Information:

Figure 14. Sample Level One Process Window

could display the program or execute it as desired. This completes the simple example.

V. IMPLEMENTATION

Although the number of potential implementations of the proposed Fourth Generation Language of the previous chapter are many, each implementation must be evaluated from the standpoint of efficiency as well as correctness. The inherent overhead of a Fourth Generation Language is tremendous since the express purpose of this new generation of languages is to remove some of the tedious tasks performed by programmers through the introduction of a vast array of defaults, menu displays and window organizations. As a result, many of the activities performed manually have been incorporated within the computer software to ease the burden on the programmer/user. Additionally, each Fourth Generation Language represents a level of abstraction above earlier programming languages and many functions/operations can be handled in a non-procedural vice procedural fashion. This can be either beneficial or detrimental to efficiency depending on how many levels of translation must occur to execute the function--a few, highly optimized translations (i.e., from high-level notation to assembly or machine language or directly to machine language) can be extremely efficient, whereas numerous translations (i.e., from high-level notation to PASCAL to assembly to machine language) may simply add to the overhead of duties the system must perform. The proposed

Fourth Generation Langauge maintains an on-line, real-time dialogue with the user and therefore requires subsecond response times so that the user will not lose his concentration nor be distracted, which would occur if he continually had to wait for the computer each time he requested a menu or manipulated one of the windows. All user input must be interpreted rather than be compiled. Once again, efficiency of implementation is mandated.

A. ARCHITECTURE

An overview of the proposed Fourth Generation Language's architecture contains the following main components: window manager, menu handler, translator, library of functions, database management system, data dictionary and database. The interconnections between these components is depicted in Figure 15.

Since the window plays such a central role in the use, development and modification of all application programs, it also serves as one of the main components of the system--centrally located between the user and the system translator. The window manager is responsible for the creation, deletion and manipulation of all windows by the user. It interprets and fills the user's request for a window and consults the data dictionary for the appropriate size and location of the window, as well as the correct edge commands to display and where to display them. The window manager determines which menus are applicable to a given window. Consistency checking

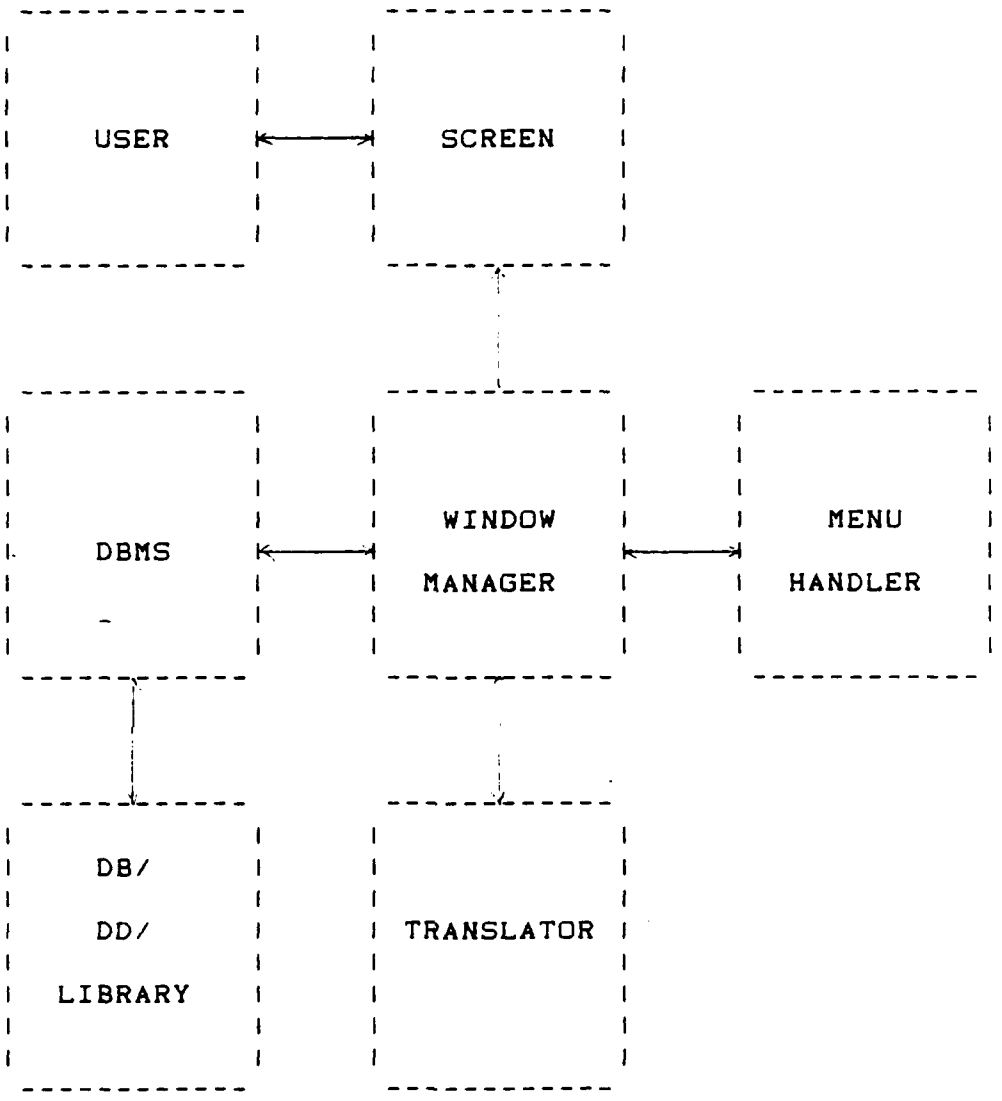


Figure 15. Textual-Graphical
System Architecture

in the following areas is performed by the window manager either alone or in conjunction with another component of the system. Before the user closes a window, the manager checks for incomplete or undefined items and lists them so that the user can be reminded, upon request, of what has not yet been completed. The window manager ensures that all input and output data from a process window (during application development) are compatible with any upper levels previously described and warns the user if not compatible. Furthermore, it checks to ensure that process window data operations are compatible as per the descriptions of data in the input and output windows. Input data that has been defined but not used in the application program or is not used in the process window it enters but never exits, is tagged for user review. Output data which is produced by a process window but not declared in an output window is also tagged.

Another important feature of the window manager is to keep track of the mode/function of the window. As the user moves the cursor between different windows--windows where the user may be performing entirely different functions (i.e., executing a program, editing/developing a program, debugging, etc.), the window manager alerts the translator as to which function is being performed in the window. With this system then, the integration of all programming tools under one environment occurs but it is spread across the various windows visible to the user. One window performs

only a single function at a time but many windows performing different functions can all be present on the computer screen at the same time and this in effect produces the integration of tools.

The menu handler interfaces with the window manager and is responsible for maintaining control of all menus presented to the user, both where and when to present them. As windows are displayed to the user, some of the standard menus or edge commands are automatically included within the window. The window manager knows which menus to include with which windows by consulting the menu handler. "Pop-up" menus periodically requested by the user or activated by user actions are also obtained by the window manager from the menu handler. The menu handler contains a list of conditions for each menu and then if conditions are met, a pointer from the menu handler to the data dictionary indicates the menu that the window manager is to include in the window.

The database management system must be fully functional, that is, allow the user to create, update, query and delete database information as well as define integrity constraints, authorized users and authorized views. In addition, the database management system must reveal the database schema upon request from an authorized user. The user must be guided through each activity, preferably through the use of graphical descriptions of the database, which appear very

easy for the user to learn and work with. Since the proposed Fourth Generation Language performs all its functions within windows, the database management system will also guide the user in his development of a query (or update, creation or deletion) within a window. An application which requires access to a database for information will develop the necessary query with the aid of the database management system and then the translator will incorporate the final query within the application program, thus the user will not be responsible for any database/program integration.

The relational database will contain an extensive data dictionary which is necessary for the consistency checking performed by the system during real-time. The data dictionary will contain all the information about all input and output data which is obtained from the input and output windows. It will contain information about all intermediate data created in process windows as well as the definitions and constraints on the data fields and tables composing the database. In addition, information regarding window initial sizes and locations and menu listings will be located in the data dictionary. The data dictionary should be available to the user to "customize" the database with his own terminology, remarks and aliases for data. The library of functions developed by the user or already present in the organization are also stored within the database. Security of the data dictionary and the library should be available to protect data and function definitions from unauthorized changes.

The translator component takes the process descriptions from the process window and translates them into code. The translator distinguishes between verbal descriptions of what a process is doing and the non-procedural functions named by the user and any procedural code he develops for the application program. The translator serves as the interface between actual code and the window manager where the high-level Fourth Generation Language is utilized in application development. In the next section, a couple of algorithms which the translator uses in translating control structure formats to code are offered.

B. ALGORITHMS

In order to show that the proposed Fourth Generation Language's features are implementable, one must not only reveal a system architecture but also develop any algorithm which may be needed in the course of translation from the high-level language to a lower level language. The architecture and algorithms themselves do not guarantee that the system proposed is a feasible one, instead they are merely the basic elements needed at the beginning of a long process to come up with an efficient and correct implementation of the proposed language. At this point, algorithms which reveal how conditional and iterative statements may be translated from the box structure of the proposed Fourth Generation Language to a Third Generation Language will be introduced.

Beginning with the conditional statement then, an algorithm utilized by the translator component will be shown. The on-line translator interacts with the user as he types information into the process window. If executable code such as mathematical statements or library defined functions are listed in the process window, the translator must keep track of which windows contain this code and which contain just verbal descriptions of the process. If the user selects the 'if-then' construct from a menu of edge commands, the window manager installs a window within the process window and this new window contains the parts of the conditional construct which the user must now fill in. Next, it is the translator's responsibility to ensure that as the conditional statement is completed by the user, only legal statements are utilized and if illegal ones are discovered, the translator must warn the user.

Once the user has completed development of his conditional statement, the translator utilizes the following algorithm to translate the user input into a PASCAL if-then statement. Note that the translator treats the user input for each part of the if-ten construct as just strings of characters.

```
BEGIN

   Initialize  (condition_o, action_o, condition_a,
               action_a, array1, array2, else_clause,
               count, no_error)

   Get_Condition (condition_o, no_error)

   Get_Action (action_o, no_error)
```

```
IF condition_o = " " AND action_o ≠ " " AND no_error
    THEN
    miss_original_cond -> error
    FALSE -> no_error

IF condition_o ≠ " " AND action_o = " " AND no_error
    THEN
    miss_original_action -> error
    FALSE -> no_error

IF condition_o = " " AND action_o = " " AND no_error
    THEN
    miss_cond_and_action -> error
    FALSE -> no_error

IF no_error THEN
    Get_Condition (cond_a, no_error)

    Get_Action (action_a, no_error)

    IF cond_a = " " AND action_a ≠ " " AND no_error
        THEN
        miss_alt_cond -> error
        FALSE -> no_error

    IF cond_a ≠ " " AND action_a = " " AND no_error
        THEN
        miss_alt_action -> error
        FALSE -> no_error

    WHILE cond_a ≠ " " AND action_a ≠ " "
                          AND no_error

        count + 1 -> count
        cond_a <- array1( count )
        action_a -> array2( count )

        Get_Condition (cond_a, no_error)

        Get_Action (action_a, no_error)

        IF cond_a = " " AND action_a ≠ " "
                        AND no_error

            THEN
            miss_alt_cond -> error
            FALSE -> no_error

        IF cond_a <> " " AND action_a = " "
                        AND no_error
             THEN
             miss_alt_action -> error
             FALSE -> no_error
```

```
        IF no_error THEN
             Get_Action (else_clause, no_error)
             Format_if_then (condition_o, action_o,
                       array1, array2, else_clause)

        ELSE
            Error_msg (error, count)

    ELSE
        Error_msg (error, count)

END
```

The procedure Get_Condition ensures that the condition in the if-then construct is either a single boolean variable or a statement with a relational operator in it, and the statement evaluates to true or false. It will also check for matching parenthesis. The procedure Get-Action checks to ensure all statements entered in the action sections of the if-then construct are legal statements for the language. Error_msg, which can be called from the main program or from each of the other two procedures, prints error messages depending on what has occurred.

For the iterative construct, basically the same activity occurs--the user requests to have a 'loop' from the menu and the window manager provides a loop window within the process window. The translator then observes what the user inputs into each part of the loop construct to ensure only legal statements are entered and warns the user when any illegal statements are entered. After the user completes the loop construct, the translator then utilizes the following algorithm.

```
BEGIN

  Initialize (body, number_lines, condition, location,
              line#, no_error)

  Get_Body (body, number_lines, no_error)

  IF body ≠ " " AND no_error THEN

      Get_Exit_Condition (condition, no_error)

      IF condition ≠ " " AND no_error THEN

          Get_Exit_Location (location, line#, no_error,
                             number_lines)

          IF line#  0 AND no_error THEN

              IF location = "Before" AND line# = 1 THEN
                   Make_While_Loop (body, condition)

              ELSEIF location = "After"
                        AND line# = number_lines THEN
                   Make_Repeat_Loop (body, condition)

              ELSE
                   Make_Loop_with_GOTO (body, condition,
                                        location, line#)

          ELSEIF no_error THEN
               negative_line# -> error
               Error_msg (error)

      ELSEIF no_error THEN
           miss_exit_condition -> error
           Error_msg (error)

  ELSEIF no_error THEN
       miss_loop-Body -> error
       Error_msg (error)

END
```

Procedure Get_Body checks for legal statements throughout the body of the loop. Get_Exit_Condition checks to ensure that the condition statement evaluates to true or false. Get_Exit_Location checks to see if the user has entered a legitimate line number. All three procedures can call the procedure Error_msg to print out any error messages.

VI. CONCLUSIONS

Although Fourth Generation Languages are no panacea, they do address some of the problems facing the computer industry today,namely--skyrocketing demand for applications development during a time of severe backlogs of requests for new applications, low programmer productivity due to the use of lower level, Third Generation Languages and needed improvement in the life cycle of large program/system development. The use of Fourth Generation Languages in program development has proven to be much more effective than Third Generation techniques, but Fourth Generation Languages have generally been less flexible in the types of applications that they can handle as compared to their predecessors. Yet to overlook the potential of Fourth Generation Languages, because of their inflexibility, seems grossly irresponsible.

Future sales of vast quantities of computers will be possible only if they can be put to work without professional programmers. Application development without programmers is perhaps one of the most important turning points in the computer revolution. With continued improvement of Fourth Generation Languages, user computing can and will flourish, but we must make these languages easier to learn and use, as well as more powerful. The combination of text and graphics enhanced through the use of windows, menus and an integrated

database management system is one approach toward achieving a more user friendly, powerful Fourth Generation Language.

LIST OF REFERENCES

Helander, G. A., "Improving System Usability for Business Professionals," IBM Systems Journal, v. 20, September 1981.

Ives, B., "Graphical User Interfaces for Business Information Systems," MIS Quarterly, v. 6, December 1982.

Martin, J., Fourth Generation Languages, Prentice-Hall, Inc., 1985.

Raeder, G., "A Survey of Current Graphical Programming Techniques," Computer, v. 18, August 1985.

Shu, N. C., Visual Programming Languages: A Dimensional Analysis, paper presented at the International Symposium on New Direcitons in Computing, Trondheim, Norway, 12 August 1985.

Stevens, W. P., "How Data Flow Can Improve Application Development Productivity," IBM Systems Journal, v. 21, June 1982.

Wu, C. T., A Unified Interface Method for Interfacing with a Database, Naval Postgraduate School, submitted for publication 1985.-

BIBLIOGRAPHY

Finzer, W. and Gould, L., "Programming By Rehearsal," Byte, v. 9, June 1984.

Purvy, R., Farrell, J., and Klose, P., "The Design of Star's Records Processing: Data Processing for the Noncomputer Professional," ACM Transactions on Office Information Systems, v. 1, January 1983.

Rowe, L. A., Fill-in-the-Form Programming, paper presented at the Proceedings of Very Large Databases, 11th, Stockholm, Sweden, 17 September 1985.

Shu, N. C., "Formal: A Forms-Oriented, Visual-Directed Application Development System," Computer, v. 18, August 1985.

Tsubotani, H., Monden, N., Tanaka, M., and Ichikawa, T., "A High Level Language Based Computing Environment to Support Production and Execution of Reliable Programs," IEEE Transactions on Software Engineering, v. 2, February 1986.

Wasserman, A. I., Pircher, P. A., and Shewmake, D. T., "Building Reliable Interactive Information Systems," IEEE Transactions on Software Engineering, v. 2, January 1986.

Zloof, M. M., "Office-by-Example: A Business Language that Unifies Data, Word Processing and Electronic Mail," IBM Systems Journal, v. 21, September 1982.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5000	2
3.	Professor Wu, Code 52Wq Naval Postgraduate School Monterey, California 93943-5000	1
4.	Professor MacLennan, Code 52Mi Naval Postgraduate School Monterey, California 93943-5000	1
5.	Captain E. L. Jacobson 14801 Sherman Way, Apt. #302 Van Nuys, California 91405	2
6.	Computer Technology Programs, Code 37 Naval Postgraduate School Monterey, California 93943-5000	1

END

10-86

DTIC